Formal Implementation of a Verification Algorithm using the B Method

Ludovic Casset

Gemplus Research Laboratory,
Av. Du Pic de Bertagne,
13881 Gémenos Cedex BP 100
Tel. +33 (0)4 42 36 40 98
Fax: +33 (0)4 42 36 55 55
Ludovic.casset@gemplus.com

Abstract: The Java security policy is implemented by security components such as the Java Virtual Machine (JVM), the API, the verifier, the loader. It is then of prime importance to ensure that the implementation of these components is in accordance with their specification. Formal methods are used to bring the mathematical proof that the implementation of these components corresponds to their specification. In this paper, the formal implementation of a verification algorithm is provided and entirely proved. It is a continuing work on the model of a byte code verifier. The specification and the implementation is realized with the B method and its associated B0 language. The work focuses on the design and the proof of loops that are a main problem to obtain a proved implementation.

Keyword: verification algorithm, B method, byte code verifier, formal methods

1 Introduction

One of the main issues when deploying applications is to provide assurance to the customer that the executions of these applications are safe. That is, their execution will not threaten the system on which they are being executed. The Java security policy defines the correct behaviour of a program and the properties that this program must hold. For example, it is not possible to forge an integer into an object reference. Java is a type-safe language.

To enforce a security policy, it is necessary to develop security components that implement this policy. In Java, these components are the Java Virtual Machine (JVM), the API, the byte code verifier, the loader. The correct construction of these components is of prime importance to ensure the security of the system. Formal methods are used to obtain the mathematical proof that the implementation corresponds to the specification and that the specification conforms to the security policy. A correct specification implies an important analysis before implementing the security functions. This analysis includes the choice of the architecture and the choice of the data structures of the model. Developing software programs this way reduces both the risk of having an implementation fault and the cost of tests.

We aim to model a general verifier for a subset of the Java byte code. In particular, we model and implement a verification algorithm using the B method. The aim of a byte code verifier is to statically check that the flow control and the data flow do not generate an error. To obtain the formal description, we worked on the formalisation of the defensive Java Virtual Machine (DJVM), a machine performing tests on an instruction just before executing it. This idea was originally developed in [Cas-99] where such a defensive machine is described and refined in order to obtain in the one hand the verifier and in the other hand the interpreter.

In this paper, we provide a full implementation of the byte code verifier modelled in a previous study and we focus on the difficult point of proving loops. The remainder of this paper is organised as follow. Related works is discussed in section 2. Section 3 introduces the project architecture and data structures used for the modelling. Section 4 describes the specification and the implementation of relevant project components. In section 5, the construction and the proof of loops are discussed, as it is one of the main issues in a B implementation. Section 6 discusses the result of this study and section 7 concludes.

2 Related Works

A large number of studies have already been led on the formal specification of the Java byte code. Cohen [Coh-97] introduces a formal specification of a Defensive Java Virtual Machine. His unfinished work is very important but does not lead to verifier.

A close work has been done by Qian [Qia-98]. The author considers a subset of Java byte code and aims to prove the type correctness at the runtime by using static constraints. Using its specification he proposes a proof of a verifier that can be deducted from its virtual machine specification. In the Bali project, Push [Pus-98] proves a part of the JVM using the prover Isabelle/HOL. Using Qian works, she gives the verifier specification and then proves its correctness. Goldberg [Gol-97] gives a formal specification of the Java byte code verifier. He simplifies his study by omitting subroutines. In a more recent work, Nipkow introduces the formal specification of the Java byte code verifier in Isabelle [Nip-00]. Its idea is to come with the generic proof of the algorithm and then instantiate it with a particular JVM.

Bertot [Ber-00] proposes a Coq formalization of a type checker. His work is based on the model introduced by Freund and Mitchell [Fre-98]. This work is very close to our, and uses the same foundation than us. He proves the one-step soundness theorem expressed in [Fre-98] and describes an effective verifier for this subset of the Java Virtual Machine language.

3 Background and Hypothesis

3.1 Generalities

The verifier that we model relies on a set of hypothesis. First, ten instructions are modeled. They represent a subset of Java byte code instructions defined by Freund et al [Fre-98] (Fig.1).

Inc adds one to the *integer* in top of stack.	**Push0** pushes *integer* on stack.
Pop removes the top element of the stack.	**If L** jumps to L or to next instruction according to the value of the *integer* L.
Istore x removes the *integer* from the top of stack and puts it into local variable x.	**Iload x** loads value from local variable x and puts it on top of stack.
Halt terminates program execution.	**New** σ allocates a new non-initialized object of type σ on the top of stack.
Init σ initializes the object of type σ on the top of stack.	**Use** σ performs an operation on a initialized object of type σ.

Fig.1. *Informal specification of the instruction subset*

They rely on three different types: **INTEGERS** for numeric types, **addr** for non-initialised reference and **addri** for initialised reference. An order relation over these different types is given in the following figure. **TOP** and **bottom** are two types added to obtain a complete lattice. This lattice is used to compare types and to compute fixpoint, that is merging types by computing the joint of the compared types.

Freund et al show the consistency and the coherence of their type system. For this reason, we use the same operational and static semantics except for the *iload* and *istore* instructions that are specialised for integers. Subroutines and exception are not taken into account in this work.

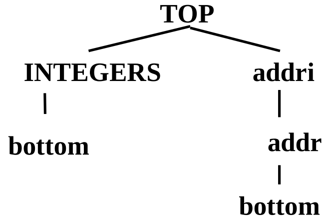

Fig.2. *The complete lattice.*

The B method is used because its theory is well suited to an industrial purpose: it covers the development process from the specification level to the implementation. This method relies on a first order logic, the predicate logic and the set theory. The development scheme of a B model includes the specification of an abstract machine, its refinements and finally its implementation. Designing an efficient B model is not a trivial task and research are currently performed to simplify it, by developing guidelines and methodology. The difficult part of this paper is to describe the proof of the implementation of the byte code verifier in B. In particular, the proof of loops is detailed in section 5.

3.2 The architecture

In this section, we briefly introduce each machine from a general view (fig.3.). This project is made of seven machines, five implementations and one refinement.
Note that all **SEES** links are not drawn in the figure for readability purposes. The different components are linked as follow:

- the *Contexte* machine is seen (**SEES**) by all components of the project,
- the *function* machine is seen (**SEES**) by the following components: *operation, operation_1, operation_I, mod_types, mod_types_I, mod_frames* and *mod_frames_i.*

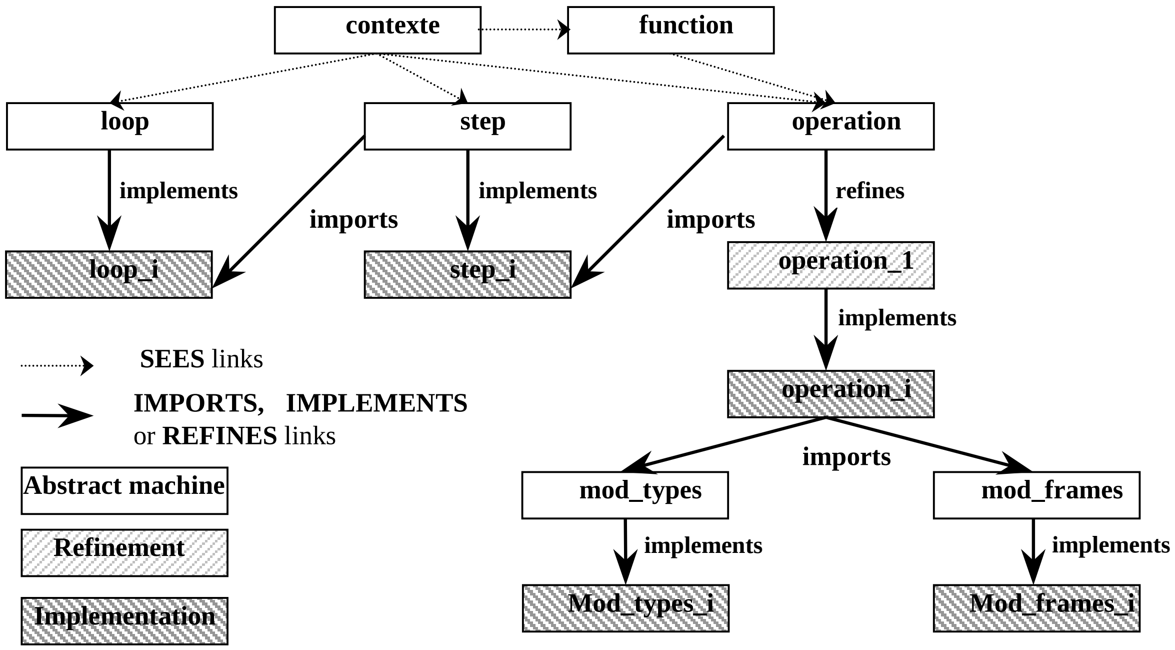

Fig.3. *The architecture of the project*

Finally, it is important to note that the *Contexte* and *function* machines are not implemented because they are only used as context machines. That is, machines containing only definitions and no operation. The short description of the machines is the following:

- *Contexte* contains the definition of data used by other machines,
- *Function* contains the definition of the complete lattice and of the comparison function,
- *Loop* is the specification of the main loop of the verifier,
- *Step* corresponds to one step of the main loop. It contains one operation that is called by the implementation of *loop,*
- *Operation* contains the specification of each instruction modelled in the project. These instructions are called by the implementation of *Step,*
- *Mod_types* and *mod_frames* are two machines modelling operations over the type stack and the frame of local variables. These machines allow implementing these data structures.

3.3 The specific context of verifiers

Verifying a program consists in ensuring its type soundness. That is, entities in the stack and in the frame are in adequacy with the program expectation, *i.e*, the program being checked. For instance, there is no non-authorised type conversion. To check that a program is correct, the verifier constructs a stack and a frame for each program point (pp). This construction is done by exploring all accessible execution paths, *i.e.* while checking the control flow. The verifier compares the different stacks to check the coherence of these stacks with the program expectations. In particular, it computes a stable stack for each pp. This computation is an iterative process. Stable means that the stack has reached a fixpoint and will not evolve anymore. It is then said that for each element, the least upper bound has been reached. While parsing the program, the verifier may have to check an instruction or an execution path several times. In fact, it checks an instruction while the fixpoint is not reached. It is the case when the type for the stack at a given pp has changed, to ensure that the new type is compatible with all execution path, the instruction and its following need to be verified once more. To make the verification process tractable, the Java byte code specification requires a stack to keep the same size for each pp. The example given in figure 4 illustrates this critical point.

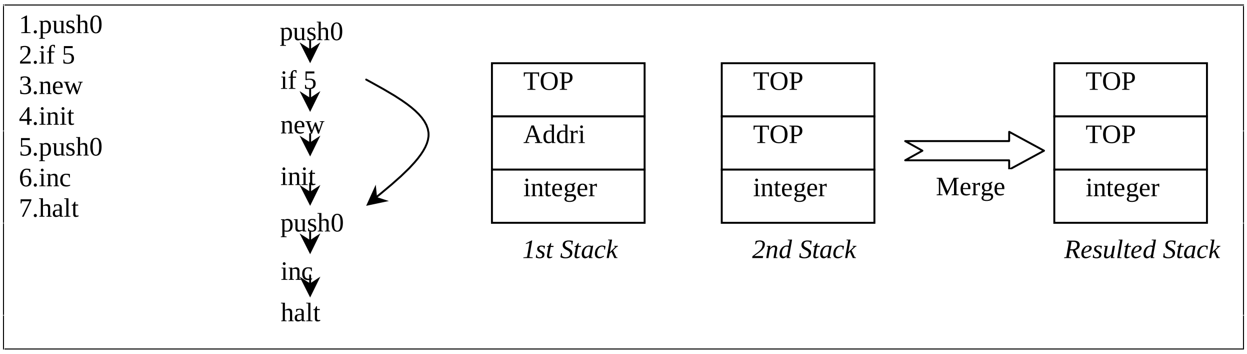

Fig.4. *A short example illustrating the construction of the static stacks by the verifier*

In this example, a simple method is shown, along with the two possible stacks while reaching the instruction 5. The first one corresponds to the program path without performing the jump (if 5). So, in the stack, there is an integer and an initialised object while reaching instruction 5. The content of the stack is given in the next figure. The second possible path corresponds to the program path performing the jump (if 5). While reaching the instruction 5, the stack only contains, in this case, an integer. Because the two stacks are different, the verifier needs to merge the stack. According to the complete lattice (fig.2), the result of the merge of the two stacks is given in the next figure.

Due to the updated stack, the verifier needs to check once more the path in order to ensure that the new type is compatible with the program expectation. In this particular case, the program is correct because the object created in the first branch is not used after instruction 5. Otherwise, an error would have occurred because the stack would not be equal.

In this context, programs are allowed to perform forward jump as in the previous example but also backward jump as described in the next example (fig.5). In this example, a jump is introduced. Proving the verification algorithm termination is one of the most difficult proof to achieve, and requires the following hypotheses:

- the lattice is complete and has a greatest element (TOP),
- the size of the stack must always be the same for each instruction, whatever execution path has been taken before.

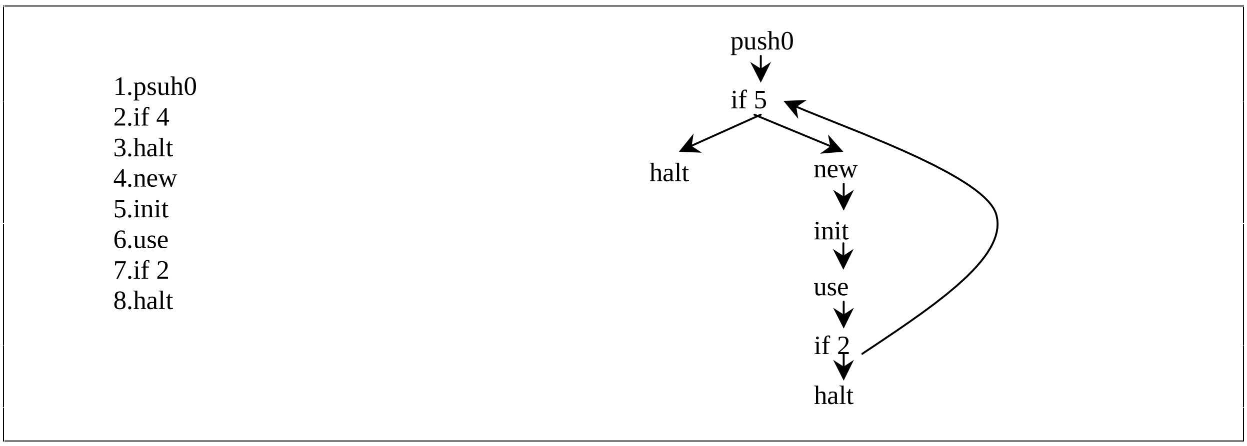

Fig.5. *A short example illustrating the loop risk for the verifier*

3.4 The verification algorithm

In our model, we consider a stack and a frame of local variables. We assume we only search a fixpoint for the stack since the two instructions that manipulate the local variables (*iload* and *istore*) only use the type *integers*. However, adding *aload* and *astore* instructions does not arise new problems. The principle of the fixpoint calculus is extended to the frame of local variables. To construct all the static stacks, *i.e.* the stack for each pp, a simple algorithm is necessary.

The proposed verification algorithm is used to compute fixpoint and to parse possible execution paths. This algorithm requires several elements: a complete lattice bounded by type *TOP* and type *Bottom*, provided with a partial order relation, and a stack (*WW*). It consists in pushing the address of the jump instruction (*if*) onto the *WW* stack, and popping them when an *halt* instruction is encountered. The stack *WW* contains all the addresses of the instructions that have to be visited and checked. Note that the jump address of the *if* instruction is stored in *WW* only if (1) this instruction is analysed for the first time and (2) the types stack for this particular pp has changed. In fact the former case can be assimilated to the latter one as initially all the types of the stack have been assigned to the *bottom* value.

At this level, the types stack and the types frame are computed for each pp. This algorithm allows checking the possible execution paths of a program but will not check dead code.

3.5 Overview of used variables

In order to determine whether an execution is correct or not, the verifier needs to have the state of the stack and of the frame of local variables at each program point. The state of the stack is characterized by its size and its content. Two variables are used to model this state: the *STtop_stack* function associates at each pp the size of the stack, the *STtypes* function of function associates at each program point the corresponding stack of types. Likewise, this stack of types is defined as a function that maps each element to its corresponding types. The state of the frame of local variables is described by the *STtypes_frames* function of function that associates at each program point the corresponding frame. The frame is modeled by a function associating each element of the frame with its type. Using function of function in the specification is very expressive but not usable in implementation. These functions of functions must be translated into tables of tables to be implementable. This last transformation is performed at the implementation level. The B definition of these three variables is given below (Fig.6).

$$
\begin{aligned}
& STtop\_stack \in 1..max\_methode \rightarrow (0..max\_stack) \\
& \wedge\ STtypes \in 1..max\_methode \rightarrow (0..max\_stack \rightarrow JTYPES) \\
& \wedge\ STtypes\_frames \in 1..max\_methode \rightarrow (0..max\_frame \rightarrow JTYPES)
\end{aligned}
$$

Fig.6. *The definition of the variables*

Several other variables are used: *pp* indicates the program point that is being checked by the verifier. The stack *WW* and the index *index* are used to specify and implement the verification algorithm. Likewise, *STtop_stack_i* allows knowing if an instruction has already been visited (TRUE) or not (FALSE). It permits to perform additional verification. The variables *error* and *stop* are respectively used to note if an error has occurred (*error* = TRUE) and if the verification is complete (*stop* = TRUE). All these variables have a concrete meaning for the verifier. In the model, other variables are introduced to ease the proof process. For instance, to compute the variant of the general loop, it is necessary to have three new variables: *OneStackLevel* and *GlobalLevel* are numeric variables ensuring that the variant is positive and strictly decreasing. To perform this task, the stack *RR* is used. It contains values to be added when no transformation of the system state is performed (in the *halt* instruction). The full description of these variables and how they are used is given in section 5.2.

4 The model

In this section, the specification of relevant project components is provided. This project deals with two different kinds of specifications: low level specifications that are close to a programming language and more abstract specification.

4.1 Low level specification

The specification introduced in this section concerns the operational part of the project, that is the *operation* machine and its refinements. This machine describes the behavior of each instruction and the evolution of the system.

The variables used to describe the system state are defined in this machine. These variables are *STtop_stack*, *STtypes*, *STtypes_frames* and *pp*. These four variables allow knowing the state of the stack and of the frame of types for each program point (*pp*). They are initialized to specific values from the earlier specification in order to ease the correspondence between machines connected by **IMPORTS** links. This initialisation (Fig.7) is performed in each refinement of this machine. The stack *WW* is empty (1). The *pp* is set to the first instruction (2). All stack lengths are set to 0 (3). No instruction except the first one has been visited (4). It is said that each element of static stacks is initialized to *bottom* type (5). So it is for the static frames (6). The program is not terminated (7). The variables used to compute the variant are set to 0 or set to empty (8). Finally, all the error variables are set to 0 or to FALSE (9).

```
index :=0                                                        (1)
   || WW:=(0..max_index)×{1}
   || pp:=1                                                      (2)
   || STtop_stack :=(1..max_methode)×{0}                         (3)
   || STtop_stack_i := ((1..max_methode)×{FALSE})<+{1↦TRUE}      (4)
   || STtypes:=(1..max_methode)×{(0..max_stack)×{bottom}}        (5)
   || STtypes_frames:=(1..max_methode)×{(0..max_frame)×{bottom}} (6)
   || stop := FALSE                                              (7)
   || OneStackLevel := (1..max_methode)×{0}                      (8)
   || GlobalLevel := 0
   || RR := ∅
   || stopnum, errornum, err := 0,0,FALSE                        (9)
```

Fig.7. *Variables initialization in Operation*

The specification of the instructions of our model indicates how the system evolves. This specification is close to the programming because it is the operational level of the instructions that is modeled. However, not all the used variables will remain in the implementation. In this case, the proof of implementation is our main concern. We detail the specification of the instruction *iload* (Fig.8). Note that all other instructions are modeled in the same way.

The prover is allowed to execute the operation only if the preconditions (1) are complete. That is if:

- the verification is not over (*Stop = FALSE*),
- no error has been detected (*err = FALSE*),
- the parameter of the operation is within the domain of the method being checked.

Then, there are the operational tests (2). They check if the call of this instruction at *pp* is correct according to the information already collected. It checks that:

- this instruction is not the last,
- there is enough room in the stack to store a new element,
- if the instruction has been visited, then the new length of the stack is the same than the previous one,
- the operation parameter exists and is within the domain of parameters,
- the type of the local variable specified by the parameter is *INTEGERS*.

If one of these verifications fails, then the verifier detects an error (5). Otherwise, the operational part is executed (3). In this part, variables are modified according to the operation being executed. In the case of the *iload* operation, the modifications are the following:

- the verifier goes to the next instruction,
- a new element is stored in the top of the stack,
- the frame remains unchanged,
- no error has been detected,
- the instruction *ii+1* is visited,
- The stack of types evolves. A fixpoint calculus is performed on all elements of the stack *STtypes* between *ii* and *ii+1*. This calculus is modeled with the **ANY WHERE** B clause (4).It computes the new stack *nt* by performing the joint between *STtypes(ii)* and *STtypes(ii+1)*. Then, the set of all static stacks is updated using the B overloading operator $<\!\!+$. The calculus of the variant is added in this clause. But the variables

concerned by this calculus will not be implemented: they are only used during the specification and disappear in the next refinement. The *GlobalLevel* variable is updated thanks to the intermediate variables *bef* and *aft*. An assertion is necessary to prove that an evolution has occurred (the types have changed).

```
error ←operation_iload(ii) =
  PRE stop = FALSE ∧ err = FALSE ∧ ii ∈ dom(methode) ∧ ii ∈ NAT1                    (1)
  THEN
     IF( ii < max_methode ∧ STtop_stack(ii)≥0 ∧ STtop_stack(ii) < max_stack          (2)
       ∧ (STtop_stack_i(ii+1)=TRUE ⇒ STtop_stack(ii+1)=STtop_stack(ii)+1)
       ∧ parametre(ii)≤max_frame ∧ parametre(ii)≥1
       ∧ STtypes_frames(ii)(parametre(ii))=INTEGERS)
     THEN
       pp := ii+1||                                                                  (3)
       STtop_stack(ii+1):=STtop_stack(ii) +1||
       STtypes_frames(ii+1):=STtypes_frames(ii)||
       error := FALSE||
       STtop_stack_i(ii+1):=TRUE ||
         ANY nt,bef,aft WHERE                                                        (4)
         nt ∈ 1..max_stack→ JTYPES
         ∧ ∀jj.(jj:(1..max_stack )-{STtop_stack(ii)+1} ⇒ nt(jj)=Meet(STtypes(ii)(jj),STtypes(ii+1)(jj)))
         ∧ ((STtop_stack(ii))+1∈ 1..max_stack ⇒ nt(STtop_stack(ii)+1)=
                 Meet(INTEGERS,STtypes(ii+1)(STtop_stack(ii)+1)))
         ∧ bef = OneStackLevel(ii+1)
         ∧ aft = ∑(yy).(yy ∈ 0..max_stack | level((STtypes(ii+1)<+nt)(yy)))
       THEN
         STtypes(ii+1):=STtypes(ii+1)<+nt||
         OneStackLevel(ii+1) := aft ||
         ASSERT bef ≤ aft
         THEN GlobalLevel := GlobalLevel - bef + aft
         END
       END
     ELSE
       error,err := TRUE,TRUE                                                        (5)
       || errornum := 1
     END
  END;
```

Fig.8. *The specification of the iload instruction*

Note that if *methode(ii)=iload* then the verifier modifies the variables for *ii+1*.

Figure 9 represents the implementation of the instruction *iload*. The refinement *operation_1* between the specification and the implementation is not depicted because it does not bring new information. An implementation of a machine requires some changes. First, all preconditions must disappear. Then, the test of a variable implies the use of intermediate variables (1), because the value of this variable is not accessible directly. The verification part of the instruction is not in a single block anymore (2). The reason for this change is that some tested variables come from imported machines and it is not possible to keep first order predicates in the tests as in specification. The operational part (3) remains unchanged except for the copy of the frame (4) and the computation of the stack of types (5) that are calls to operation from imported machines (*mod_types* and *mod_frames*). Finally, note that the order of the instruction has been kept to ease the match with its specification.

Note that the implementation and the proof of the different loops used in the operational part are not provided in this section to keep the model understandable. The implementation and the proof of the loops are described in section 5.

```
error ←operation_iload(ii) =
  VAR zz,tt,thof,varstack_i,varstack,zzs IN                                   (1)
      tt:=parametre(ii);
      zz:=STtop_stack(ii);
      IF( ii < max_methode ∧ zz≥0 ∧ zz < max_stack ∧ tt≤max_frame ∧ tt≥1)        (2)
      THEN
      thof← read_frames(ii,tt);
      IF( thof=INTEGERS)                                                        (2)
      THEN
      varstack:=STtop_stack(ii+1);
      varstack_i:=STtop_stack_i(ii+1);
      zzs:=zz+1;
      IF(varstack_i=TRUE ∧ varstack≠zzs)                                        (2)
      THEN error:=TRUE
      ELSE
              pp := ii+1;                                                       (3)
              STtop_stack(ii+1):=zz +1;
              dup_frames(ii,ii+1);                                              (4)
              error := FALSE;
              STtop_stack_i(ii+1):=TRUE ;
              meet_types_bis(ii,ii+1,zz+1,INTEGERS)                             (5)
      END
      ELSE error := TRUE
      END
      ELSE error := TRUE
      END
  END;
```

Fig.9. Implementation of the instruction iload in operation_I

4.2 The main theorem

The main theorem is another kind of specification. It is an abstract specification at the opposite of the previous one, which is more concrete. It is very simple to express for human but rather hard to prove by the machine. The main theorem we attempt to prove simply states that the verification process must end and therefore, does not loop indefinitely.

Theorem: *The verification process ends*

It is critical for a byte code verifier to end its verification. A result is expected and must be true or false depending if the program to check is correct or not. If the verifier does not terminate, it means that there is not a finite answer to that question. In fact, it is not able to state whether the program is correct or not.

Proving this theorem means that the construction and the check performed by the verifier are not leading to a blocking situation. Showing a program termination and bringing the proof of its termination is not quite easy when the main loop is not bounded. The B expression of the previous theorem is depicted in figure 10. It simply specifies that the result of the verification must be **TRUE** or **FALSE** and excludes any other result.

```
static_error ←  treechecking =
  PRE Cold = TRUE
  THEN
      Cold := FALSE ||
      IF(methode(max_methode)≠halt) THEN static_error:=TRUE
      ELSIF(max_methode≠1) THEN static_error:∈ BOOL
      ELSE static_error:=FALSE
      END
  END
```

Fig.10. *The specification of the main theorem*

Note that this specification eliminates the limit cases: if the program is not ending with a *halt* instruction, the operation raises an error. Likewise, if the program contains only one instruction, it must be a *halt* or it is an error.

Finally, the last thing to note about this specification is the use of the variable *cold*. This abstract variable limits the call of the operation *treechecking*. It is necessary since the proof of the loop relies on initialisation performed in other machines. So, if the operation is called twice, variables will not be initialised for the second time and the verification will provide wrong results. The variable *cold* avoids this risk.
The implementation of this machine is not difficult thanks to the chosen architecture. The following figure depicts this implementation. The main issue was to find a suitable variant. This choice is discussed in the section 5.2. The algorithm ends when there is no more instruction to check (*converge* = *TRUE*) or if it has detected an error (*local_error* = *TRUE*). The loop body is made of a call of one instruction. The operation *OneStep* is defined in the machine *Step* described below in this section.

```
static_error ← treechecking = BEGIN
  VAR value1, local_error,converge
  IN
   local_error:=FALSE;
   converge := FALSE;
   value1:=methode(max_methode);
   IF(value1≠halt) THEN local_error:=TRUE
   ELSIF(max_methode ≤ 1) THEN
   WHILE (converge=FALSE & local_error=FALSE) DO
       local_error, converge ← OneStep
   INVARIANT
       local_error: BOOL & converge : BOOL & MyVariant : Integers &
       MyVariant ≤ (max_methode + (max_methode +1)×(3×max_methode×(max_stack+1)) + 2) &
       converge = Stop & local_error = Erreur
   VARIANT
       (max_methode + (max_methode + 1)×(3×max_methode×(max_stack+1)) + 2 + 1) − (MyVariant)
    END
    END;
    static_error := local_error
   END
  END
```

Fig.11. *The implementation of the general loop of the verifier*

The *Step* machine corresponds to one step of the loop. That is, the verification of one instruction. The specification is only concerned by the evolution of the variant. It specifies that after a step of the loop, the new variant, *nv*, is strictly bigger than the previous one, *MyVariant* (Fig.12.). The operation returns two Boolean values *Erreur* and *Stop* that are used to stop the loop. Therefore, the operation has four preconditions. Two are related to the limit cases identified in the loop specification. The latter two preconditions forbid the execution of this operation if an error has been detected or the algorithm ends.

```
Local_error, converge ← OneStep =
PRE max_methode ≠ 1 ∧ methode(max_methode) = halt ∧ Stop = FALSE ∧ Erreur = FALSE
THEN
ANY nv,ns,ne WHERE
        nv ∈ INTEGER
        ∧ nv ≤(max_methode + (max_methode + 1)×(3×max_methode×(max_stack+1)) + 2)
        ∧ MyVariant < nv
        ∧ ns ∈ BOOL ∧ ne ∈ BOOL
THEN
        MyVariant := nv ||
        local_error,Erreur := ne,ne ||
        converge,Stop := ns,ns
END
END
```

Fig.12. *The B specification of the operation* OneStep

Finally the link with the operational part (the *operation* machine) appears in this implementation (Fig.13.). It is an enumeration of all cases with the B clause **IF ELSIF**. Note that a B restriction forced us to split the *if* operation in two different operation. This restriction concerns the generalised substitution that can modify a variable only once in an operation.

```
local_error, converge ← OneStep =
VAR value,spc IN
spc ← getpc;
value:=methode(spc);
IF( value=halt) THEN local_error ← operation_halt
ELSIF(value=inc) THEN local_error ← operation_inc(spc)
ELSIF(value=pop) THEN local_error ← operation_pop(spc)
ELSIF(value=push0) THEN local_error ← operation_push0(spc)
ELSIF(value= iload) THEN local_error ← operation_iload(spc)
ELSIF(value= istore) THEN local_error ← operation_istore(spc)
ELSIF(value= new) THEN local_error ← operation_new(spc)
ELSIF(value=init) THEN local_error ← operation_init(spc)
ELSIF(value= use) THEN local_error ← operation_use(spc)
ELSIF(value= if) THEN
        local_error ← operation_if(spc);
        IF(local_error=FALSE)
        THEN local_error← operation_iff(spc)
        END
ELSE local_error:=TRUE
END;
converge←getconvergence
END
```

Fig.13. *The implementation of the OneStep Operation*

5 Loop implementation

This section describes the implementation of two different types of loop that have been encountered in the project. We focus on the loop implementation as it is the most important issue when implementing a B model. Few industrial guidelines and methodologies are provided to help the developer to design and prove a loop using the B method. The first loop is used very often as it copies the content of table into another table. The second one is more specific and has the particularity of not being bounded: the number of step is unknown. Both are difficult to prove.

When designing a loop several issues arise. First of all the higher specification must correspond to the result expected at the end of the loop. In the particular case of the main loop of the verifier, the specification is the B expression of the theorem introduced in section 4.2 (Fig.10.).

A B loop is an implementation and is composed of three different elements [Abr-96] that are detailed hereafter. The first element is the body of the loop. It contains the different actions performed during the execution of the loop. The second is the variant, a quantity evolving during the loop. The variant helps proving the loop termination. In practice, the variant must always be positive and strictly decreasing. Finally, the last element required for the loop is the invariant, a set of rules that always have to be held by the variables manipulated within the loop.

The proof of a loop implies the generation of three different kinds of Proof Obligations (POs):

- POs for the loop initialisation: mathematical formulae that need to be verified before the loop starts,
- POs for the loop execution: mathematical formulae that need to be verified during the loop execution,
- POs for the loop termination: mathematical formulae that need to be verified after the loop termination.

The expression of these POs can be complicated by the Atelier B toolkit as the name of the different variables are completed with some id ($2, $7777; etc) to distinguish them at the different levels for the loop. So the reading of a PO is rather difficult since it is long and complicated, and so is the proof.

5.1 A bounded loop

The specification of a copy of a table is simple. The following figure depicts the *meet_types_bis* operation of the *mod_types* machine. This operation performs a copy of the stack *src* into the stack *dest* except for the element *vm*. The joint of types is computed directly on the elements during the copy except for the *vm* element. The type of this element is computed between the previous element *STtypes(dest)(vm)* and the *value* passed as parameter of the operation.

```
meet_types_bis(src,dest,vm,value)=
  PRE src∈1..max_methode ∧ dest∈1..max_methode ∧ dest≠src ∧ vm∈1..max_stack ∧ value∈ JTYPES
  THEN
   ANY nt WHERE                                                                         (1)
    nt∈ 1..max_stack → JTYPES
    ∧ ∀jj.(jj:(1..max_stack)-{vm} ⇒ nt(jj)=Meet(STtypes(src)(jj),STtypes(dest)(jj)))
    ∧ (vm∈1..max_stack ⇒nt(vm)=Meet(value,STtypes(dest)(vm)))
   THEN
    STtypes(dest):=STtypes(dest)<+nt                                                    (2)
   END
  END ;
```

Fig.14. *A loop specification in mod_types*

The use of the **ANY** structure permits to compute independently the new stack (1) and then to perform the substitution (2). The implementation of this operation uses the standard libraries provided with the Atelier B toolkit and more precisely the *L_ARRAY1_RANGE* machine. This machine proposes a set of operation allowing translating function of function into an implementation of table of table. It is necessary since only tables of tables are implementable. As the proof of the *L_ARRAY1_RANGE* machine is provided, it is interesting to reuse it. However, it proposes standard services such as copy, initialisation of tables but, the implementation of the previous operation requires a particular implementation as described in the following figure.

```
meet_types_bis(src,dest,vm,value)= VAR mi IN
    mi := 0;
    WHILE mi < max_stack DO
    VAR v1, v2, vr IN
        mi := mi + 1;
        v1 ← types.VAL_ARR_RGE(src, mi);
        v2 ← types.VAL_ARR_RGE(dest, mi);
        IF(mi≠vm)
        THEN
                vr := Meet(v1,v2);
                types.STR_ARR_RGE(dest, mi, vr)
        ELSE
                vr:= Meet(value,v2);
                types.STR_ARR_RGE(dest, mi, vr)
        END
    END
    INVARIANT                                                                    (1)
     mi ∈ 0..max_stack ∧                                                         (2)
     types.arr_rge ∈ 1..max_methode→(0..max_stack→JTYPES) ∧
     ∀xx.(xx ∈ 1..max_methode ∧ xx ≠ dest ⇒ types.arr_rge(xx) = STtypes(xx)) ∧    (3)
     ∃nti.(nti ∈ 1..mi → JTYPES ∧                                                 (4)
        ∀kk.(kk : (1..mi)-{vm} ⇒ nti(kk)=Meet(STtypes(src)(kk),STtypes(dest)(kk))) ∧
        (vm∈1..mi ⇒ nti(vm)=Meet(value,STtypes(dest)(vm)) )
       ∧ types.arr_rge(dest) = STtypes(dest) <+ nti)
    VARIANT                                                                      (5)
     MAXINT-mi
    END
   END;
```

Fig.15. The implementation of the operation *meet_types_bis*

In the case of a copy of a table the variant is very easy to find. The table counter iterated by the loop is a good one (5). The developer can easily managed to build the counter with the property of being always positive and strictly decreasing. The difficult part is the invariant (1). It is composed of three parts: the types of the variables used within the loop (2), the general invariant of the loop (3) and the most delicate to find, a predicate providing the state of all variables at each step of the loop (4). This last predicate has the particularity to be likely as the predicate used in the abstraction of the operation in the **ANY WHERE** clause. The difference is that it has now an existential form and the variable domain is not he same. In the loop, this domain evolves until it reaches the same set that the one given in specification. At the end, the result obtained is the one specified in the abstraction. This invariant is proved to be true while entering the loop, during its execution and at its end.

5.2 The verifier's main loop

This loop is the most difficult one to design and to prove in the whole project. In fact, it represents the implementation of the main theorem of the project. The difficulty appears in the construction of the variant part of the loop because the number of step is unknown. So a simple counter as described in the previous section is not sufficient. The search of the variant has two objectives: first find a correct one that is well suited to the loop needs and second, find one that can ease the proof of the loop.

It has been chosen to attribute a value to each type. The function *level* defined in the *Contexte* machine gives the correspondence between the types and their value. The state of the stack is calculated for each program point. Each stack is bounded by *max_stack* and contains *max_stack+1* elements, all initialized to *bottom*. The variables *STtypes* can be depicted as a matrix of types of *(max_stack+1)×max_methode* elements (Fig.16). By using the *level* function, this matrix of types can be translated into a matrix of values. It is then possible to perform the sum on all elements of each stack. This sum is called *OneStepLevel*. Finally a positive quantity over all stacks is computed, *GlobalLevel*.

bottom	bottom	bottom
bottom	TOP	TOP
addr	addr	addri
integers	integers	integers

0	0	0
0	3	3
2	2	1
2	2	2

OneStepLevel(1)=4
OneStepLevel(2)=7
OneStepLevel(3)=6

GlobalLevel = 17

Fig.16. *Correspondence between the matrix of types and GlobalLevel*

Initially, all elements are set to *bottom* so the value of *GlobalLevel* is 0 (*level(bottom)=0*). To simplify the calculus, *OneStackLevel* is used as an intermediate value. The definition of these two variables, *OneStackLevel* and *GlobalLevel,* is given below.

$$OneStackLevel(xx) = \sum_{yy=0}^{\max\_stack} level(STtypes(xx)(yy))$$

$$GlobalLevel = \sum_{xx=1}^{\max\_methode} OneStackLevel(xx) - \sum_{uu=1}^{index} RR(uu)$$

The *xx* parameter of *OneStackLevel* is the program point where this value is computed. *STtypes* represents the set of the static stacks of the program. *OneStackLevel(xx)* represents the numeric value associated to the stack at a given program point. *GlobalLevel* represents the numeric value of all the static stacks. However, some value can be removed. In fact, the variant is a quantity that strictly decreases. So it must always evolves. When an *if* instruction is being verified, two modifications are done to *STtypes*.

- the calculus of the stack for *pp+1*, *STtypes(pp+1)*,
- the calculus of the stack for the jump destination, *STtypes(parametre(pp))* where *parametre(pp)* is the jump address.

This last calculus is performed and stored in a stack named *RR*, a twin stack of the stack *WW* used to implement the verification algorithm. When the verifier encounter a *halt* instruction, it removes the last value of *RR* and add it to *GlobalLevel*. *GlobalLevel* keeps to evolve even for the *halt* instruction. *GlobalLevel* is an increasing value. This property is provided by the fact that while computing the fixpoint, the type of a variable can only increase in the lattice but stays bounded by the upper type *TOP*. This important property allows first ensuring the increasing of *GlobalLevel* and second ensuring the existence of the fixpoint and the n the termination of the algorithm. *RR* contains all the values of modified stacks associated to jumps already encountered. All these values are positives.

GlobalLevel alone is not a correct variant because in some cases it does not evolve. When an error occurs or when the algorithm ends, the variant has to evolve but no variable describing the system state is modified. The solution we propose is to associate a numeric value to the error variable and the termination variable. So, when an error is detected (*error = TRUE*), the new variable *errornum* is set to 1. Otherwise, the value is 0. Likewise, when the algorithm ends, the state of the stack does not evolve anymore. The variable *Stop* indicates if the verifier has finished (TRUE) or not (FALSE). Its associated numeric variable *stopnum* is set to 0 when the verifier has not finished and to 1 when the algorithm ends.

By grouping all these variables together and using the fact that the *pp* evolves too, a strictly decreasing and positive quantity can be extracted: pp **+** (max_methode+1) × GlobalLevel **+** stopnum **+** errornum

This quantity is bounded by :

max_methode **+** (*max_methode* **+** 1) × 3 × *max_methode* × (*max_stack* + 1) + 1 + 1.

In fact, we have $pp \leq max\_methode$, $GlobalLevel \leq 3 \times max\_methode \times (max\_stack + 1)$, $stopnum \leq 1$ and $errornum \leq 1$. Finally if the variable *MyVariant* has the following value

MyVariant = pp + (max_methode + 1) × GlobalLevel + stopnum + errornum,

Then, $max\_methode + (max\_methode + 1) \times 3 \times max\_methode \times (max\_stack + 1) + 2 + 1 - MyVariant$

is a strictly decreasing and positive quantity. This quantity is the variant of the general loop of the verifier. All these variables are manipulated by the operational part of the project, *operation*. Furthermore, they do not have a concrete meaning and are not implemented. The figure 8 depicts the use of these variables in the *iload* instruction.

There are some general remarks concerning the construction of this variant. There is no automatic procedure to find a loop variant. The general idea is to find a strictly decreasing and positive quantity. It seems difficult to provide simple guidelines allowing to find such a complex variant. However, once the idea of the variant is emerging, the Atelier B interactive prover can help to refine this idea until obtaining a suitable variant. For example, the interactive prover determined the choice of the order of the terms of the variant. In fact, writing a formula in a particular order helps the prover to understand it and to manipulate it efficiently. Likewise, note that the value bounding the variant keeps the same order than the variant, term to term. Finally, the invariant of this loop was entirely found thanks to the interactive prover: the prover asks the developer to prove certain formulas that indicate that some parts of the invariant are missing. Constructing the variant and the invariant using this method is very interesting, as the result is the entire automatic proof of the specification and the implementation of the *loop* machine.

6 Results

This project is entirely proved. We proved the implementation of each machine except *Contexte* and *Function* that only contain constant declarations. Therefore, we bring the proof that the verification algorithm used in this model ends. The global results of the project are summarized in figure 17.

COMPONENT	TC	POG	Obv	nPO	nUn	%Pr	B0C
Contexte	OK	OK	1	0	0	100	OK
Functions	OK	OK	1	3	0	100	OK
Loop	OK	OK	5	0	0	100	OK
Loop_i	OK	OK	30	8	0	100	OK
Mod_frames	OK	OK	5	6	0	100	OK
Mod_frames_i	OK	OK	31	46	0	100	OK
Mod_types	OK	OK	7	12	0	100	OK
Mod_types_i	OK	OK	54	97	0	100	OK
Operation_1	OK	OK	1187	378	0	100	OK
Operationi	OK	OK	793	467	0	100	OK
Operation	OK	OK	527	248	0	100	OK
Step	OK	OK	7	1	0	100	OK
Step_i	OK	OK	170	109	0	100	OK
TOTAL	OK	OK	2818	1375	0	100	OK

Figure.17. *The global result for the project*

Over the 1375 proof obligations (POs), 246 have to be manually proven. This means a 82% rate of automatic proof, which is a high rate and a good result. Over the remaining 246 P0s, some of them, that the case for the *operation* machine and its refinements, are similar, which ease the manual proof process. The difficult part of the implementation is the loop implementation and we succeed to prove loops thanks to a specific architecture and experience acquired in this domain. Not only the most complex loop is proved but also the proof of this loopis entirely automatic.

7 Conclusion and Future works

In conclusion, this study demonstrates the capabilities of the B method to model a verification algorithm and to implement it. We have identified critical points that make the implementation difficult: first, a B implementation requires only basic types. So, the data structures used in the specification must be translated into simpler ones. Second, the design and the proof of loops in B are not simple. The model brings to the fore solutions for both critical points. This paper mainly focuses on solution to the design and the proof of loops.

The architecture described in figure 3 has an important impact on the proof of the project and more precisely for the proof of the loop of the verification algorithm. The issue arises by the loop is that the implementation mixes up the complexity of the operational part with the specification. In fact, the specification must match the operational part but only at the end of the loop. This architecture, by specifying the loop body in a separate machine, simplifies the proofs generated by the Atelier B toolkit and splits the difficulties in several machines (*loop*, *step* and *operation*). The given architecture has two main advantages: First, it provides a simpler higher specification of the loop. This specification is likely the B expression of the theorem the developer wants to prove. It is more readable and understandable for human and allows refining and implementing the loop body distinctly from the loop. Second, the POs generated by this architecture are simpler for the developer and even for the automatic prover. A confusion generally occurs when the automatic prover attempt to prove the loop. Using this architecture, there is no more confusion. The POs generated have the same meanings but they are simpler and in some cases, they are automatically proved. This architecture can easily be generalized to prove loop. However it requires an important analysis of the design and on what the developer wants to prove. It is useful to prove some theorem as shown in this project or in [Req-00], but it is not necessary for loop used to copy element.

Finally, the formal implementation of a verification algorithm is provided. We have proved the termination of the algorithm through the proof of its implementation and the implementation proof of the operational part. The extension of this work to a more general or a complete Java byte code verifier is possible for the operational part. That is, an extension of the *operation* machine with all others instruction. However, exceptions and subroutines are still not taken into account.

Future works have two different goals. The first one concerns this model. In this study, the proof of the implementation of a verification algorithm is provided. We are now aiming to link this work with the work performed in [Cas-99]. In fact, the starting point is the model of the defensive machine that leads to the formal specification of the byte code verifier. We have already modeled the lower and the higher part. The work is now to link both parts. The second goal is the development of an embedded verifier for smart card. This work is based on the work on the Proof-Carrying Code (PCC) by Necula [Nec-97] and its adaptation to Java Card by Rose [Ros-98]. The difficulty is to adapt a verification algorithm to the limited resources available in a smart card. But, we now have experience in the model and the proof of a verification algorithm.

8 Acknowledgement

We would like to thank Denis Sabatier from Stéria for his help on the design and the proof of the model.

9 References

[Abr-96] J. R. Abrial, *The B Book. Assigning Programs to Meanings*, Cambridge University Press 1996.

[Ber-00] Y. Bertot, *A Coq formalization of a Type Checker for Object Initialization in the Java Virtual Machine*, Research Report, INRIA Sophia Antipolis, to appear.

[Cas-99] L. Casset, J-L. Lanet, *A Formal specification of the Java Byte code Semantics using the B method*, Proceedings of the ECOOP'99 workshop on Formal Techniques for Java Programs, June 1999.
http://www.informatik.fernuni-hagen.de/pi5/publications.htrml

[Coh-96] Cohen, *Defensive Java Virtual Machine Specification*
http ://www.cli.com/software/djvm

[Fre-98] S. N. Freund, J. C. Mitchell *A type System for Object Initialization in the Java Byte code Language* In. Proc. Conf. On Object-Oriented Programming, Systems, Languages, and Applications, pages 310-328. ACM Press 1998.
http ://theory.standford.edu/~freunds.

[Nec-97] G. Necula, *Proof Carryng Code*, 24th Symposium on Principles of Programming Languages, POPL'97, pp 106-119, Paris, January 1997.

[Nip-00] T. Nipkow, *Verified Byte code Verifiers*, Fakultät für Informatik, Technische Universität München, 2000.
http://www.in.tum.de/~nipkow

[Qia-98] Z. Qian, *Least Types for Memory Locations in Java Byte code*, Kestrel Institute, Tech. Report, 1998.

[Req-00] A. Requet, L. Casset, G. Grimaud, *Application of the B Method to the Proof of a Type Verification Algorithm*, HASE 2000, Albuquerque, New Mexico, USA, October 2000.

[Ros-98] E. Rose, K. Rose, *Lightweight byte code Verification*, In OOPSLA'98 Workshop Formal Underpinnigs of Java, 1998.